

FORTH, APL, LPB, I.A., etc...

SEPTEMBRE 1986

SOUS LE CAPOT DU 83-STANDARD: l'éditeur

BASES DE DONNEES ET SYSTEMES EXPERTS: suite

EDITORIAL

Enfin de la couleur. Jedi se veut plus dense, plus vivant, plus animé. Et il y réussit très bien avec votre aide. Réaliser un mensuel imprimé tous les mois et dont vous ne nous faites que des éloges , méritait une touche de couleur. Il nous semblait important de marquer nos deux années d'existence par une amélioration de la maquette du journal. Si la typographie intérieure est parfois un peu sommaire, nous nous rattrapons sur l'aspect de la couverture. Il n'est pas toujours facile de trouver des sujets de décoration. L'option la plus simple aurait consisté à choisir un sujet austère et figé. Mais comme son contenu, JEDI évolue et tient à le faire savoir.

Dans ce numéro, certains articles sont des réponses à diverses questions qui nous ont été posées (tracé d'arcs de cercles, par exemple), rédigées sous forme d'articles assez courts. Si vous-même avez des trucs et des astuces de programmes rédigés en quelques lignes, faites en part aux autres adhérents. Souvent, ce peut être le départ d'une idée de programme plus ambitieu pour le lecteur. Le partage de la connaissance donne toujours des retombées dont les bénéfices peuvent être une collaboration sur un même sujet ou un gain de temps sur un problème auquel on n'en a que peu à consacrer. Et dans JEDI, les idées les délirantes sont les bienvenues (voir BLAISE), sans limite de domaine, de système ou de langage.

SOMMAIRE

ASSOCIATION JEDI 8, rue Poirier de Narçay 75014 PARIS

Tel: (1) 45.42.88.90 (de 10h à 18h)

FORTH LE BLAISE 3e épisode par Eric AUBOURG

Pour mettre en pratique les deux premiers épisodes, j'ai défini un langage inspiré de PASCAL, le BLAISE.

Il est assez aisé de vérifier sur les diagrammes syntaxiques que nos deux lois sont respectées. Pour écrire l'analyseur, j'ai choisi le FORTH: ce langage s'y prête très bien. Pour pouvoir utiliser les routines du noyau, j'ai imposé la condition que tous les identificateurs soient séparés par au moins un espace (généralement les compilateurs acceptent que l'on accole un signe de ponctuation et un identificateur, ceux-ci ne comportant alors que des caractères alphanumériques).

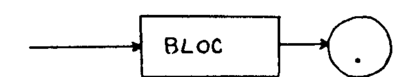
Le programme source doit être tapé dans un bloc buffer, l'usage de ' -- ' étant admis. L'analyse est lancée par n PROGRAMME, n étant le numéro du premier écran du programme (comme LOAD).

Les codes d'erreur sont les suivants:

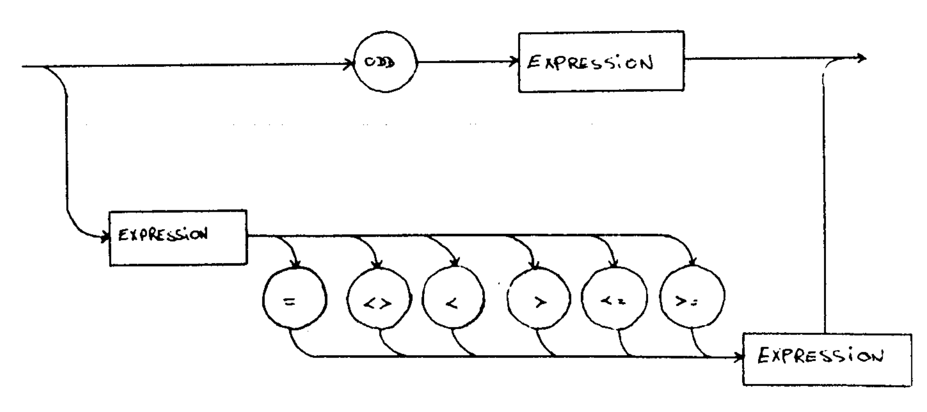
1	utiliser = au lieu de :=
2	nombre attendu après =
3	= attendu
4	identificateur attendu après CONST, VAR, PROCEDURE
5	, ou ; attendu
9	. attendu
12	variable attendue
13	:= attendu
14	identificateur de procédure attendu
16	then attendu
17	; ou end attendu
18	do attendu
20	opértateur relationnel attendu
22	) attendu
25	(attendu

Dans le prochain numéro, nous aborderons la génération du code objet, et nous écrirons un véritable compilateur pour le BLAISE (Ndlr: tout ceci est ... Achement BLAISE ... comme disait le pote, celui avec sa salopette rayée ...).

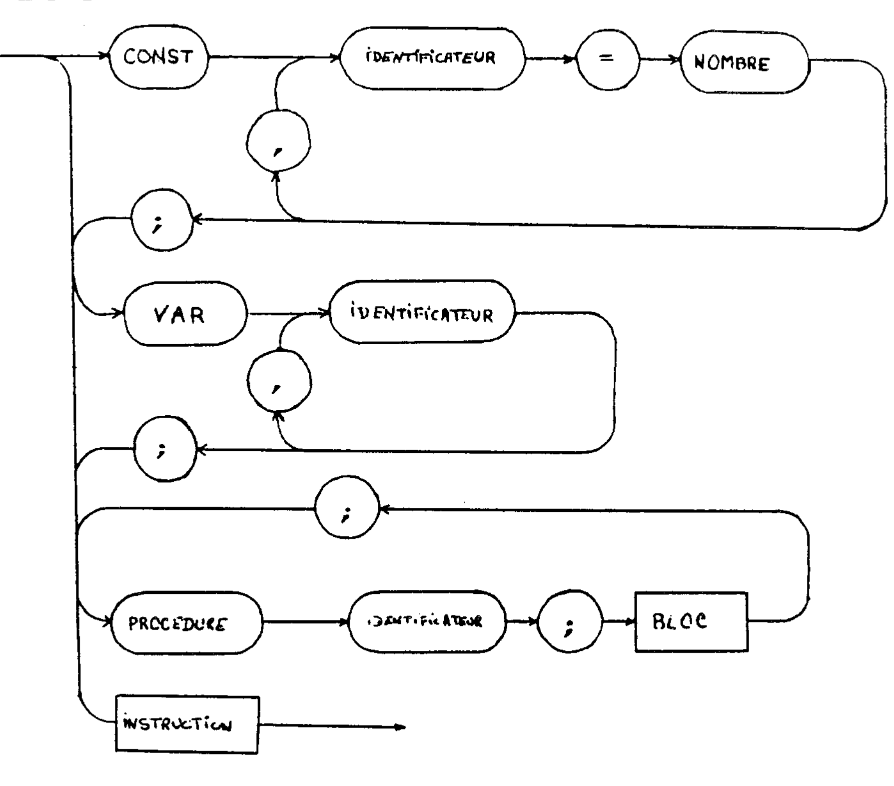
PROGRAMME



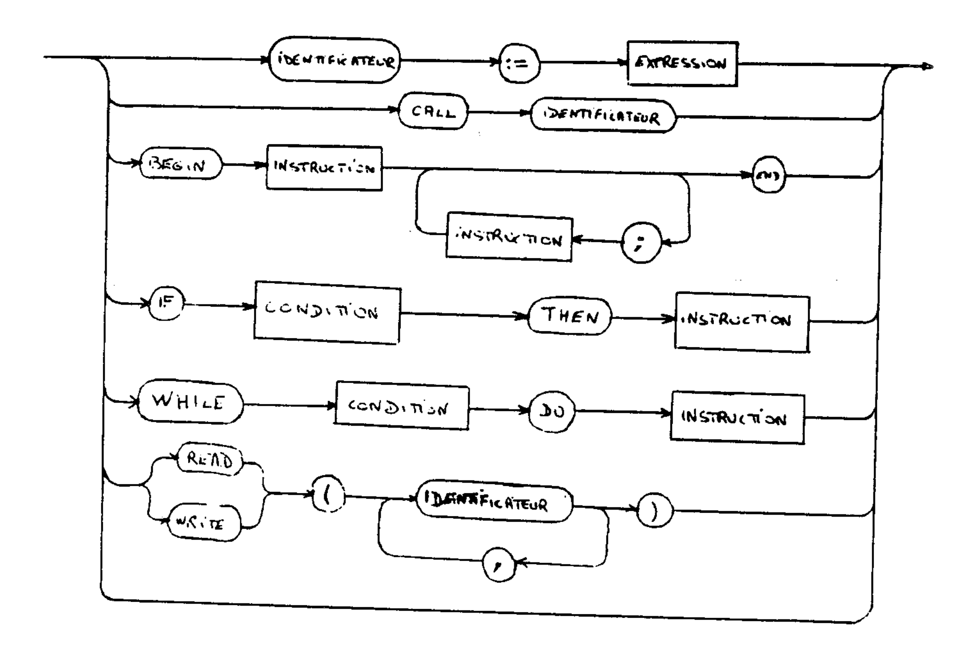
CONDITION



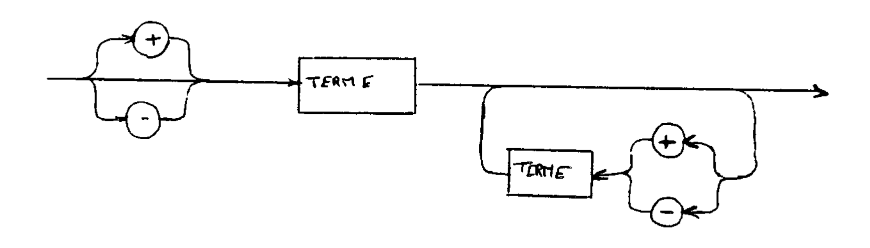
BLOC



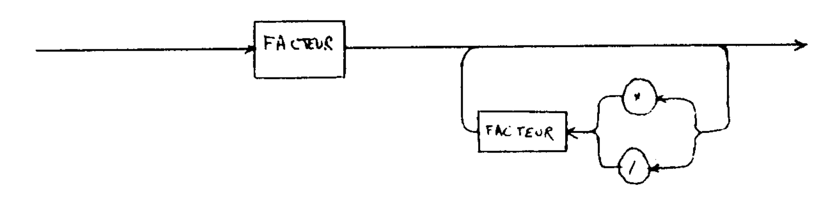
INSTRUCTION



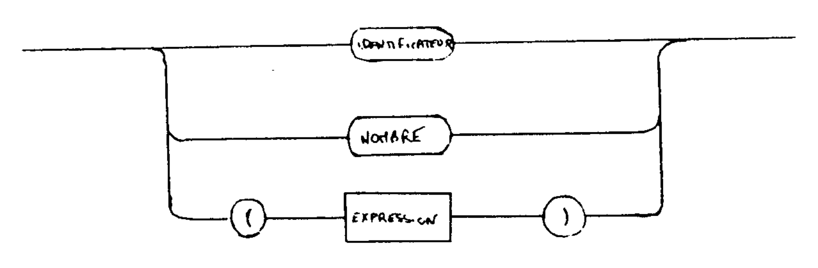
EXPRESSION



TERME



FACTEUR




```
SCR # 9
( *** ANALYSEUR BLAISE 1 *** )

VOCABULARY MOTRESERVES MOTRESERVES DEFINITIONS
 1 CONSTANT BEGIN     2 CONSTANT CALL
 3 CONSTANT CONST     4 CONSTANT DO
 5 CONSTANT END       6 CONSTANT IF
 7 CONSTANT ODD       8 CONSTANT PROCEDURE
 9 CONSTANT THEN     1Ø CONSTANT VAR
11 CONSTANT WHILE    12 CONSTANT READ
13 CONSTANT WRITE    15 CONSTANT )
17 CONSTANT :=       18 CONSTANT +
19 CONSTANT -        2Ø CONSTANT *
21 CONSTANT /        22 CONSTANT =
23 CONSTANT >        24 CONSTANT <
25 CONSTANT <>       26 CONSTANT <=
27 CONSTANT >=       28 CONSTANT ;          -->

SCR # 1Ø
( *** ANALYSEUR BLAISE 2 *** )
14 CONSTANT (        16 CONSTANT ,
29 CONSTANT .        3Ø CONSTANT -->
FORTH DEFINITIONS

' MOTRESERVES CONSTANT LIMITVOC
: ERREUR
  ." ERREUR #" . BLK @ ." BLOC #" . IN @ 64 / ." LIGNE #"
  . CR HERE COUNT TYPE QUIT ;
: GETSYM
  DROP -FIND
  IF DROP CFA DUP LIMITVOC > ELSE Ø DUP THEN
  IF EXECUTE DUP 3Ø =
    IF [COMPILE] --> [ SMUDGE ] GETSYM [ SMUDGE ] THEN

        -->

SCR # 11
( *** ANALYSEUR BLAISE 3 *** )





 ELSE
 DROP HERE NUMBER DROP DROP 31
 THEN ;
Ø
: FACTEUR
DUP 3Ø > OVER 34 < AND IF GETSYM ELSE
DUP 34 = IF 21 ERREUR ELSE
DUP 14 = IF GETSYM [ HERE SP@ 14 + ! Ø , ]
15 - IF 22 ERREUR THEN Ø GETSYM ELSE
23 ERREUR THEN THEN THEN ;                -->
```

ERIC AUBOURG — 26 MAI 1986

```
SCR # 12
( *** ANALYSEUR BLAISE 4 *** )
: TERME
BEGIN
  FACTEUR DUP 2Ø = OVER 21 = OR
WHILE
  GETSYM
REPEAT ;
: EXPRESSION
DUP 18 = OVER 19 = OR
IF GETSYM THEN
BEGIN
  TERME DUP 18 = OVER 19 = OR
WHILE
  GETSYM
REPEAT ;
' EXPRESSION CFA SWAP !                         -->


SCR # 13
( *** ANALYSEUR BLAISE 5 *** )
: CONDITION
DUP 7 =
IF GETSYM EXPRESSION
ELSE EXPRESSION DUP 21 > OVER 28 < AND
 IF GETSYM EXPRESSION
 ELSE 2Ø ERREUR
THEN THEN ;
: INSTRUCTION [ SMUDGE ]
DUP CASE
32 OF GETSYM DUP 17 =
      IF GETSYM EXPRESSION
      ELSE 13 ERREUR THEN ENDOF
33 OF 12 ERREUR ENDOF
34 OF 12 ERREUR ENDOF
 2 OF GETSYM DUP 34 =                                -->


SCR # 14
( *** ANALYSEUR BLAISE 6 *** )
      IF GETSYM
      ELSE 14 ERREUR THEN ENDOF
 6 OF GETSYM CONDITION DUP 9 =
      IF GETSYM INSTRUCTION
      ELSE 16 ERREUR THEN ENDOF
11 OF GETSYM CONDITION DUP 4 =
      IF GETSYM INSTRUCTION
      ELSE 18 ERREUR THEN ENDOF
  1 OF GETSYM BEGIN INSTRUCTION DUP 28 =
       WHILE GETSYM REPEAT DUP 5 -
       IF 17 ERREUR THEN
       GETSYM ENDOF
ENDCASE DUP 12 = OVER 13 = OR IF GETSYM DUP 14 -
IF 25 ERREUR THEN GETSYM BEGIN DUP 32 = OVER 33 = OR Ø=
IF 12 ERREUR THEN GETSYM DUP 16 = WHILE GETSYM   -->

SCR # 15
( *** ANALYSEUR BLAISE 7 *** )
REPEAT
DUP 15 - IF 22 ERREUR THEN GETSYM
THEN ; SMUDGE
: BLOC [ SMUDGE ]
DUP 3 =
IF BEGIN
    33 CONSTANT GETSYM DUP 22 -
    IF 3 ERREUR THEN GETSYM DUP 31 -
    IF 2 ERREUR THEN GETSYM DUP 16 -
  UNTIL


  DUP 28 - IF 5 ERREUR THEN GETSYM THEN
DUP 1Ø =
IF BEGIN                                        -->
```

```
SCR # 16
( *** ANALYSEUR BLAISE 8 *** )
     32 CONSTANT GETSYM DUP 16 -
     UNTIL
    DUP 28 - IF 5 ERREUR THEN
    GETSYM THEN
BEGIN
 DUP 8 =
WHILE
34 CONSTANT GETSYM 28 - IF 5 ERREUR THEN
LATEST Ø GETSYM BLOC >R CURRENT @ ! R>
DUP 28 - IF 5 ERREUR THEN GETSYM
REPEAT INSTRUCTION ; SMUDGE
: PROGRAMME
MOTRESERVES DEFINITIONS CR                  -->


SCR # 17
( *** ANALYSEUR BLAISE 9 *** )
BLK @ >R IN @ >R B/SCR * BLK ! Ø IN !
Ø GETSYM BLOC 29 - IF 9 ERREUR THEN
[COMPILE] FORTH DEFINITIONS
R> IN ! R> BLK ! ." ANALYSE TERMINEE " ;
```

ERIC AUBOURG   26 MAI 1986

APL PREMIERE APPROCHE

par F. ESPINASSE

Au début des années 60, le mathématicien K.E. IVERSON a ressenti le besoin d'élaborer un langage concis, permettant de décrire les états et l'évolution des processus physiques se déroulant à l'intérieur des ordinateurs. Il élabora un système de notation qui porte son nom et dont l'ensemble constiteu la base du langage APL. Implémenté sur ordinateur, ce langage fut très vite doté d'un ensemble très puissant de primitives mathématiques, particulièrement apte à traiter les variables multidimensionnelles (vecteurs, matrices, tableaux).

UN LANGAGE CONVERSATIONNEL

Apparu à une époque où la règle générale en informatique était le traitement par lots (batch processing), le langage APL mettait en oeuvre un nouveau type de relation homme-machine, le dialogue par écran-clavier, tel que nous le connaissons maintenant sur les terminaux et les micro-ordinateurs.

Le langage APL est donc dès l'origine un langage essentiellement conversationnel (interactif).

UN SYSTEME COMPLET

Le langage (ou le système) APL a été dès l'origine conçu comme un système de développement complet, incluant un éditeur de fonctions simple mais efficace, et un système d'archivage de programmes et de données par sauvegarde des espaces de travail.

Il est encore possible dans certaines limites de travailler entièrement sous APL sans faire appel aux fonctionnalités du DOS. Ce type de fonctionnement a été repris par la suite sur les micro ordinateurs travaillant exclusivement sous BASIC par exemple.

DES REGLES SIMPLES ET UNIVERSELLES

Les règles du langage APL reposent sur les notations d'IVERSON qui constituent un ensemble logique et cohérent en comparaison avec l'anarchie des notations mathématiques classiques. Par exemple, il y a deux types d'opérateurs:

- l'opérateur monadique qui admet un argument à droite.

- l'opérateur dyadique (ou diadique) qui admet un argument gauche et un argument droite. Cette règle ne souffre pas d'exception.

Factorielle 2 s'écrira: ! 2

l'opérateur factorielle "!" étant monadique. De même, il n'y a pas de priorité entre les opérateurs. Toutes les opérations sont effectuées dans l'ordre, de la droite vers la gauche. Ainsi, le résultat de

2 x 3 + 6 est 18

Au début, ça surprend (NDLR: sauf les FORTHiens chevronés) nos esprits rompus (ou corrompus) aux notations mathématiques classiques. On s'y fait encore piéger de temps à autre après des années de pratique. Mais c'est un élément de simplicité et d'universalité des règles d'APL.

UN LANGAGE PORTABLE

Grace à cette simplicité et cette universalité, le langage APL repose sur une norme bien établie et reconnue, dont les diverses implémentations s'écartent très peu. On n'y retrouve pas la floraison de dialectes que connaissent d'autres langages. Si l'on ajoute à celà que les programmes APL sont du code source que l'on peut stocker sous forme de fichiers de caractères, on voit que la portabilité de l'APL est très bonne.

Suite en page 16

Au fil des leçons précédentes, nous avons cherché à illustrer les principales caractéristiques qui font l'originalité du langage L.P.B. (langage pseudo-basic).

Difficile de classer celui-ci dans la jungle informatique actuelle : espèce en voie de disparition avant d'avoir vraiment vécu, grand fauve encore au biberon, ou simple ectoplasme ?

Tout réfléchi, la meilleure comparaison qui s'impose, c'est celle de la chauve-souris : au langage évolué (BASIC par exemple), on emprunte l'allure externe, la facilité de manipulation et de mise au point, et au langage-machine (en pratique l'assembleur) la vitesse d'exécution, la compacité du code et la possibilité de faire faire à un micro-ordinateur exactement ce que l'on veut.

Cette performance est obtenue grâce à un compilateur dénommé BALCOM qui a la propriété d'être lui-même écrit en langage L.P.B. et d'exister sur une grande variété de machines différentes, puisqu'un dialecte particulier à chaque micro-processeur a été défini, reflétant le jeu de registres et d'opérations propre à chacun d'eux.

Comme beaucoup de nos lecteurs, le langage L.P.B. a fini par succomber à la tentation représentée par l'univers des ordinateurs compatibles avec IBM-PC, qu'il a entrepris de coloniser, embarquant avec lui tout le savoir accumulé lors de ses séjours chez les autres types de microprocesseurs (Z80, 6502 et 6809).

Partant de l'idée qu'une exploration raisonnée doit commencer par le commencement, nous invitons fermement ceux qui partent à la découverte de ce type de machines de méditer préalablement notre premier exemple qui montre de façon simple et rigoureuse comment, en respectant l'architecture logicielle d'un IBM-PC, on peut :

- exploiter le système général d'interruptions logicielles
- surveiller l'état du clvaier
- gérer l'écran monochrome en mode texte
- convertir du code ASCII en hexadécimal

A l'attention des néophytes, et avant de vous inviter à découvrir tous les détails de l'exemple, rappelons que :

- Le micro-processeur 80-88 travaille avec des adresses de 20 bits formées par combinaison entre un registre de segment (16 bits) qu'il décale de 4 bits vers la gauche avant de lui ajouter une adresse relative (16 bits).

- l'acès normal à l'écran passe par l'interruption logicielle INT 16 fournie par le BIOS (Basic Input-Output System), à moins que l'on ne préfère écrire directement dans la zone RAM qui commence à l'adresse absolue &HB0000.

- l'accès au clavier utilise normalement l'interruption logicielle INT 22, mais on peut aussi consulter directement l'état des bascules principales telles que CAPS LOCK et NUM LOCK, qui se trouve dans l'octet RAM d'adresse &H417.

Nous avons aussi désassemblé et commenté le code exécutable produit par BALCOM à partir du texte L.P.B., de façon à bien montrer le détail des mécanismes.

Nous accueillerons bien volontiers à l'avenir dans cette même rubrique tout petit programme de même style pouvant contribuer à mieux faire connaître le fonctionnement interne d'un compatible IBM-PC.

- - - - - - - - - - - - -

→

Voici un exemple de petit programme capable de visualiser en permanence sur votre écran l'état des bascules NUM LOCK et CAPS LOCK du clavier :

```
200 '================= DEMO.LPB ==== 80-88 ====================
1000 ' Ceci est un programme-source de démonstration du LPB
1010 ' Il illustre la manière de gérer les entrée-sortie
1020 ' et vous permettra, par évolution progressive
1030 ' de développer vos propres applications.
1040 ' ----------------------------------------------------------
1050 ' Cet exemple fonctionne de la façon suivante :
1060 ' Il reproduit en haut à gauche de votre écran
1070 ' l'état des bascules CAPS LOCK et NUM LOCK,
1080 ' et affiche la valeur hexadécimale de ce que vous tapez
1090 ' jusqu'à ce que vous ayez tapé ESCAPE pour sortir.
1100 ' ----------------------------------------------------------
1110 ' Pour faire fonctionner l'exemple, vous devez :
1120 '  1/ le compiler en tapant simplement RUN
1130 '  2/ le lancer en tapant DEMO=&HB000:CALL DEMO
1140 ' ==========================================================
1150 ORG &HB000  'Pour que le module soit exécutable immédiatement
1160 GOTO PRINCIPAL
1170 :
1180 LABEL TITRE
1190  DATA "Analyse des codes internes du clavier d'un IBM-PC,"
1191  DATA " par (c) LOGICIA :             ",0
1200 :
1210 PROCEDURE INTSCR(AH):INT &H10:RETURN   ' Gestion de l'écran
1220 PROCEDURE INTKBD(AH):INT &H16:RETURN   ' Gestion du clavier
1230 PROCEDURE INTLPT(AH):INT &H17:RETURN   ' Gestion de l'imprimante
1240 :
1250 PROCEDURE EFFACECRAN                   ' Effface l'écran
1260  INTSCR(0)
1270 RETURN
1280 :
1290 PROCEDURE ENVOIE(AL)                   '  Envoie AL vers l'écran
1300   BH=0
1310   INTSCR(14)
1320 RETURN
1330 :
1340 PROCEDURE MESSAGE(SI)                  ' Affiche un message à l'écran
1350  AL=(SI+):IF AL<>0 THEN ENVOIE(AL):GOTO MESSAGE
1360 RETURN
1370 :
1380 PROCEDURE TESTCLAVIER                  ' Teste la présence d'un carac
1390  INTKBD(1)
1400 RETURN
1410 :
1420 PROCEDURE ATTEND GIVING AL             ' Attend le prochain caractère
1430  INTKBD(0)
1440 RETURN
1450 :
1460 PROCEDURE SURVEILLEBASCULES
1470  DEFB ETATDUCLAVIER SYN &H17
1480  PUSH A,DS,ES
1490   DS=A=&H40                  ' Zone des paramètres système
1500   ES=A=&HB009                ' Adresse de la page video monochr
1510   AH=10:FLD=0                ' Pour incrémenter
1520   DI=10                      ' Adresse video = ES*16+DI=&HB009A
```

Et voici ce que donne, après compilation par BALCOM, ce petit programme, désassemblé avec l'utilitaire DUMPIBM fourni avec BALCOM sur la même disquette :

```
.PA
                    * -------- désassemblage de DEMO.BIN -----------
B000  E9E600      jmp #B0E9              ; GOTO PRINCIPAL

B003  41...       DB  'Analyse des codes  internes du clavier '
B029  64...       DB  'd'un IBM-PC, par LOGICIA :              '
B053  00          DB  0
B054  CD10        int #10                ; PROCEDURE INTSCR(AH)
B056  C3          ret

B057  CD16        int #16                ; PROCEDURE INTKBD(AH)
B059  C3          ret

B05A  CD17        int #17                ; PROCEDUER INTLPT(AH)
B05C  C3          ret

B05D  B400        mov ah,=#00            ; PROCEDURE EFFACECRAN
B05F  EBF2FF      <<call #B054>>
B062  C3          ret

B063  B700        mov bh,=#00            ; PROCEDURE ENVOIE(AL)
B065  B40E        mov ah,=#0E
B067  E8EAFF      <<call #B054>>
B06A  C3          ret

B06B  AC          lodsb                  ; PROCEDURE MESSAGE(SI)
B06C  3C00        cmp al,=#00
B06E  7406        jz #B076
B070  E8F0FF      <<call #B063>>
B073  E9F5FF      jmp #B06B

B076  C3          ret

B077  B401        mov ah,=#01            ; PROCEDURE TESTCLAVIER
B079  E8DBFF      <<call #B057>>
B07C  C3          ret

B07D  B400        mov ah,=#00            ; PROCEDURE ATTEND ...
B07F  E8D5FF      <<call #B057>>         ;        ... GIVING AL
B082  C3          ret

B083  50          push ax                ; PROCEDURE SURVEILLEBASCULES
B084  1E          push ds
B085  06          push es
B086  B84000      mov ax,=#0040
B089  8ED8        mov ds,ax
B08B  B809B0      mov ax,=#B009          ; ES=&HB009
B08E  8EC0        mov es,ax
B090  B40A        mov ah,=#0A
B092  FC          cld
B093  BF0A00      mov di,=#000A
B096  A01700      mov al,(#0017)         ; AL=ETATDUCLAVIER
B099  2420        and al,=#20
B09B  740B        jz #B0A8
B09D  B04E        mov al,=#4E            ; N
B09F  AB          stosw
B0A0  B055        mov al,=#55            ; U
```

```
1530   AL=ETATDUCLAVIER AND &H20         ' Bascule NUM LOCK
1540   IF <> THEN AL="N":(DI+)=A:AL="U":(DI+)=A:AL="M":(DI+)=A
             ELSE AL="D":(DI+)=A:AL="E":(DI+)=A:AL="P":(DI+)=A
1550   DI=6                              ' Adresse video = ES*16+DI=&HB0096
1560   AL=ETATDUCLAVIER AND &H40         ' Bascule CAPS LOCK
1570   IF <> THEN AL="K" ELSE AL=" "
1580   (DI+)=A
1590  POP ES,DS,A
1600 RETURN
1610 :
1620 PROCEDURE HEXA1(AL)                 ' Affiche un digit hexadécimal AL
1630  IF AL>9 THEN AL=AL+55 ELSE AL=AL+"0"
1640  ENVOIE(AL)
1650 RETURN
1660 :
1670 PROCEDURE HEXA(AL)                  ' Affiche l'octet AL en hexadécima
1680  PUSH A,A
1690   HEXA1(AL SRL 4)
1700   POP A
1710   HEXA1(AL AND 15)
1720  POP A
1730 RETURN
1740 :
1760 CONSTANT ESCAPE = 27
1770 :
1780 LABEL PRINCIPAL
1790  EFFACECRAN:MESSAGE(@TITRE):FOR BL=80:ENVOIE("="):NEXT BL
1800 :
1810 LABEL BOUCLE
1820  SURVEILLEBASCULES
1830  TESTCLAVIER :IF = THEN BOUCLE
1840  AL=ATTEND:IF AL=ESCAPE THEN FAR RETURN
1850  HEXA(AL):IF AL=0 THEN HEXA(AH)
1860  ENVOIE(" ")
1870 GOTO BOUCLE
1880 END PRINCIPAL '-----------------------------------------------

B107  E873FF     <<call #B07D>>       ; AL=ATTEND
B10A  3C1B       cmp al,=#1B          ; IF AL=ESCAPE ...
B10C  7501       jnz #B10F            ;  ... THEN ...
B10E  CB         ret                  ;   .... FAR RETURN

B10F  E8C2FF     <<call #B0D4>>       ; HEXA(AL)
B112  3C00       cmp al,=#00          ; IF AL=0...
B114  7505       jnz #B11B            ; ...THEN...
B116  8AC4       mov al,ah
B118  E8B9FF     <<call #B0D4>>       ; ... HEXA(AH)
B11B  B020       mov al,=#20
B11D  E843FF     <<call #B063>>       ; ENVOIE(" ")
B120  E9DAFF     jmp #B0FD            ; GOTO BOUCLE

           * --------------- fin de DEMO.BIN ---------------
```

```
B0A2  AB         stosw
B0A3  B04D       mov al,=#4D          ; M
B0A5  AB         stosw
B0A6  EB09       jmp #B0B1

B0A8  B044       mov al,=#44          ; D
B0AA  AB         stosw
B0AB  B045       mov al,=#45          ; E
B0AD  AB         stosw
B0AE  B050       mov al,=#50          ; P
B0B0  AB         stosw
B0B1  BF0600     mov di,=#0006
B0B4  A01700     mov al,(#0017)       ; AL=ETATDUCLAVIER
B0B7  2440       and al,=#40
B0B9  7404       jz #B0BF
B0BB  B04B       mov al,=#4B          ; K
B0BD  EB02       jmp #B0C1            ;

B0BF  B020       mov al,=#20          ; espace
B0C1  AB         stosw
B0C2  07         pop es
B0C3  1F         pop ds
B0C4  58         pop ax
B0C5  C3         ret

B0C6  3C09       cmp al,=#09          ; PROCEDURE HEXA1(AL)
B0C8  7E04       jle #B0CE
B0CA  0437       add al,=#37
B0CC  EB02       jmp #B0D0

B0CE  0430       add al,=#30          ; "0"
B0D0  E890FF     <<call #B063>>
B0D3  C3         ret

B0D4  50         push ax              ; PROCEDURE HEXA(AL)
B0D5  50         push ax
B0D6  D0E8       shr al,1
B0D8  D0E8       shr al,1
B0DA  D0E8       shr al,1
B0DC  D0E8       shr al,1             ; AL=AL SRL 4
B0DE  E8E5FF     <<call #B0C6>>       ; premier digit
B0E1  58         pop ax
B0E2  240F       and al,=#0F          ; AL=AL AND 15
B0E4  E8DFFF     <<call #B0C6>>       ; deuxième digit
B0E7  58         pop ax
B0E8  C3         ret

B0E9  E871FF     <<call #B05D>>       ; LABEL PRINCIPAL
B0EC  BE03B0     mov si,=#B003
B0EF  E879FF     <<call #B06B>>       ; MESSAGE(@TITRE)
B0F2  B350       mov bl,=#50          ; FOR BL=80
B0F4  B03D       mov al,=#3D
B0F6  E86AFF     <<call #B063>>       ;  ENVOIE("=")
B0F9  FECB       dec bl
B0FB  75F7       jnz #B0F4            ; NEXT BL
                                      ; LABEL BOUCLE
B0FD  E883FF     <<call #B083>>       ; SUREVEILEBASCULES
B100  E874FF     <<call #B077>>       ; TESTCLAVIER
B103  7502       jnz #B107            ; IF = .... .
B105  EBF6       jmp #B0FD            ;  ... THEN BOUCLE
```

FORTH Routines générales

par Gérard SOULA

REFERENCES

Difficuté de programmation: moyenne
Catégorié: utilitaire
Difficulté d'exercice: facile

L'EXERCICE:

Il s'agit de se constituer quelques utilitaires que l'on pourra utiliser dans de nombreux autres programmes.

LE PROGRAMME

PAGE efface l'écran et dépend de votre système. La définition donnée est valable sur le COMMODORE 64.

RND (n --- n') générateur de nombre aléatoire n' compris entre 0 et n-1.

RANDOM (n --- n') nombre aléatoire entre 1 et n.

COMPTEUR variable servant de compteur.

ZERO remise à zéro du compteur.

AJOUTE incrémente de 1 le compteur.

COMBIEN lit la valeur contenue dans le compteur.

WITHIN (n inf sup --- f) teste si un nombre n se trouve dans l'intervalle inf sup-1 et donne 1 si oui, 0 si non.

```
10 10 20 WITHIN   donne 1
15 10 20 WITHIN   donne 1
19 10 20 WITHIN   donne 1
 9 10 20 WITHIN   donne 0
20 10 20 WITHIN   donne 0
```

On utilise la pile retour et le test logique AND

O/N (--- f) attend une touche au clavier. Ignore si ce n'est pas O ou N, rend 0 si c'est N et 1 si c'est O.

ENCORE? (--- f) est destiné à être utilisé dans une boucle BEGIN..UNTIL. On aura donc 1 pour s'arrêter, 0 pour continuer.

```
( ROUTINES GENERALES I )

: PAGE  ( - )   147 EMIT ;

VARIABLE (RND)   HERE (RND) !

: RANDOMISE  ( n-n )  (RND) @  31421 * 6927 +
   DUP (RND) ! ;
: RND ( n-n' )   RANDOMISE U* SWAP DROP ;
: RANDOM  ( n-n' )  1+  RND  1 MAX ;


: WITHIN  ( n inf sup - f )
   >R 1- OVER <
   SWAP R>  <  AND ;
```

```
( ROUTINES GENERALES II )

: O/N   ( - f )
   BEGIN                       ( debut da la boucle )
     KEY                       ( attend une touche  )
     DUP 79 =                  ( est-ce O ? )
     IF 1 SWAP                 ( si oui on met 1 et on swape le flag)
     ELSE 78 = IF 0 1          ( sinon si c'est N on met 0 )
               ELSE 0          ( si c'est ni O ni N on recommence )
               THEN
     THEN
   UNTIL ;

: ENCORE?  ( - f )  ." Voulez vous continuer? O/N "
   O/N  0= ;
```

```
( ROUTINES GENERALES III )

VARIABLE COMPTEUR

: ZERO      0 COMPTEUR ! ;
: AJOUTE    1 COMPTEUR +! ;
: COMBIEN   COMPTEUR @ ;
```

* LES MOTS "NONCE" *

par W. Baden (FIG, dec. 84).

Le problème des "mots se définissant eux-mêmes" revient souvent en Forth, mais est rarement reconnu pour ce qu'il est.

C'est un paradoxe du Forth Standard que l'on puisse définir un mot <nomX> que l'on utilise ensuite pour définir un mot <nom>, que <nomX> ne sera plus jamais utilisé, mais que l'on ne peut définir <nom> directement.

Par exemple, supposons que nous voulions un mot, CTR, incrémentant de 1 le nombre qu'il contient à chaque appel. Une première méthode consiste à définir deux mots:

```
CREATE   TALLY  0 ,

: CTR ( --- n )  TALLY DUP @  1 ROT +! ;
```

Mais on peut aussi définir un mot de définition et l'utiliser une fois:

```
: COUNTER   CREATE 0 ,  DOES>  DUP @   1 ROT +! ;

COUNTER   CTR
```

Bien sûr si nous pouvons définir un mot pouvant définir <nom> nous devrions être capable de définir <nom> directement. Henry Laxen propose une solution basée sur une propriété accidentelle d'une méthode particulière d'implantation. Cette approche, qui est strictement interdite par le Standard, ne fonctionne pas avec les codes à chainage direct, ni les codes compilés, ni avec des schémas de segmentation mémoire; et elle est assez vicieuse à comprendre.

```
: CTR  DOES>  DUP @  1 ROT +! ;  CTR  0 IS CTR
```

Il est cependant plus productif de reconnaitre le problème, et d'en chercher une solution (qui existe). Dans ce sens, une extension au (prochain) standard est proposé.

```
:DOES>        -- adr
              -- sys    (compilation)
```

définit le comportement à l'exécution d'un mot créé par un mot de définition de haut niveau. Il est utilisé sous la forme

```
<create> <nom> .... :DOES> .... ;
```

où <create> est CREATE ou tout mot utilisateur de définition utilisant CREATE et <nom> est le dernier mot défini dans le vocabulaire de compilation.

Il commence la définition de la partie "exécution" de <nom>. Lorsque <nom> est exécuté, l'adresse du champ paramètre de <nom> est empilée puis la séquence de mots entre :DOES> et ; est exécutée.

La définition de CTR est alors:

```
CREATE  CTR 0 ,  :DOES>  DUP @  1 ROT +! ;
```

→

":DOES>" possède une affinité certaine avec "DOES>"; les ":" suggèrent qu'une séquence compilée de mots commence, se terminant par un ";". Le ">" nous rappelle que quelque chose est empilé, et les ":"

au début du mot plutôt qu'à la fin nous signalent que ce n'est pas un mot de définition.

Si DOES> peut être implanté, il en est de même de :DOES>. Pour le modèle F83, voici comment procéder:

Si DOES-SIZE, nombre d'octets de code machine compilés par DOES>, vaut 3, on peut faire les définitions suivantes:

```
' FORTH  (un mot "DOES" typique)  @ COUNT

      CONSTANT  DOES-OP

      @  CONSTANT  DOES-ADDR

: :PATCH>   HERE SWAP !
      DOES-OP C,  DOES-ADDR ,  !CSP ]  ;
```

Ceci suppose que la machine utilise un adressage absolu pour effectuer le saut vers le code généré par DOES>. Si elle utilise des sauts relatifs, comme sur le 8086:

```
' FORTH  @  COUNT

      CONSTANT  DOES-OP

      LENGTH + CONSTANT DOES-ADDR

: :PATCH>   HERE SWAP !
      DOES-OP C,  DOES-ADDR HERE 2+ - ,  !CSP ] ;
```

Si DOES-SIZE vaut 4, remplacez "COUNT" par "LENGTH" et "C," par ",".

```
: LATEST   ( --- nfa )
      CURRENT @ #THREADS  LARGEST NIP L>NAME ;

: :DOES>  LATEST DUP  LAST !  NAME> HIDE :PATCH> ;
```

LATEST est un vieil ami de FIG-Forth et contient le champ nom de la dernière définition du vocabulaire de compilation.

Deux mots utiles pour ce concept sont:

```
: VALUE  CREATE  ,  DOES>  @ ;

: DEFER  CREATE [']  CRASH  ,  DOES>  PERFORM ;
```

où CRASH affiche un message d'erreur approprié (et exécute ABORT) si une autre valeur n'a pas été assignée à <nom>.

CTR devient maintenant:

```
Ø VALUE CTR  :DOES>  DUP @  1 ROT +! ;
```

Voici d'autres exemples (extraits de Laxen et Perry):

```
DEFER PAGE  :DOES> PERFORM
      1 PAGE# +!   #LINE OFF   #OUT OFF ;
```

(de même pour AT et DARK).

→

PERFORM et OFF ont pour définition:

```
: PERFORM  (adr -- qcq )
     @ EXECUTE ;

: OFF      ( adr -- )
     Ø SWAP ! ;
```

La définition de SWITCH du méta-compilateur de Laxen-Perry est confuse. Mais si on fait d'abord:

```
: EXCH  ( adr1,adr2 -- : échange de leurs valeurs )
     2DUP @ >R  @ SWAP !  R> SWAP ! ;
```

on obtient ensuite:

```
CREATE SWITCH
     CONTEXT @  ,  CURRENT @  ,
:DOES>  DUP  CONTEXT  EXCH
     2+ CURRENT EXCH ;
```

:DOES> peut être utilisé dans d'autres définitions de Laxen-Perry.

```
100 VALUE MS  ( n -- : délai d'env. n mS )
:DOES> @ SWAP  Ø ?DO  DUP  Ø DO LOOP  LOOP DROP ;
```

Un générateur de nombres pseudo-aléatoires est comme CTR:

```
HERE VALUE RANDOM
:DOES> DUP @  31429 *  1+  DUP  ROT ! ;
```

:DOES> est bien adapté aux tables mathématiques, surtout si un traitement supplémentaire est requis:

```
CREATE SINUS Ø , ... ( valeurs de la table )
:DOES> ... ;
```

Voici un mot empilant la valeur (Ø ou 1) d'un bit donné d'un octet:

```
CREATE BIT  ( octet,bit# -- Ø/1 )
1 C,  2 C,  4 C,  8 C,  16 C,  32 C,  64 C,  128 C,
:DOES>  + C@ AND Ø= NOT ABS ;
```

Si vous soupçonnez un sens caché au mot :PATCH>, vous avez raison. Vous ne pouvez l'utiliser dans un programme standard, mais il peut être utile pour "déboguer" dans un environnement approprié.

Par exemple:

```
: IT  ." ONE." ;
: TRY ." THIS IS " IT ; TRY
' IT :PATCH> DROP ." TWO." ; TRY
' IT :PATCH> >R ." ANOTHER " ; TRY
```

et TRY affiche:

```
THIS IS ONE.
THIS IS TWO.
THIS IS ANOTHER ONE.
```

---

traduction A. J., sept. 86.

■

III
BASE DE DONNEES
ET MECANISMES DE DEDUCTION

Cette approche est plus orientée Bases de Données que les précédentes, elle consiste à augmenter les possibilités des Systèmes de Gestion de Bases de Données par l'adjonction de mécanismes de déduction. Ces mécanismes peuvent être implantés de diverses manières, celles-ci sont présentées au moyen des exemples qui suivent.

BASE DE DONNEES RELATIONNELLLES
EXPERIMENTALE
BASEE SUR LA LOGIQUE
(Jack MINKER)

C'est une Base de Données Relationnelle basée sur la logique, élaborée par Jack MINKER à l'Université du MARYLAND.

La logique est utilisée pour représenter les connaissances, elle forme une base mathématique du raisonnement sur les données et du maintien de l'intégrité de la base de données. Ce maintien de l'intégrité de la base est indispensable si l'on veut dériver de nouveaux faits explicitement à partir de la base.

Ce système est utilisé pour gérer de grosses bases de données ainsi que pour résoudre des preuves de théorèmes.

* Les requêtes sont énoncées sous forme de formules bien formées du calcul des prédicats.

* Les connaissances du système sont stockées sous forme de réseaux sémantiques. Ceux-ci sont mémorisés dans la base de données explicites.

* Les axiomes permettant de dériver de nouveaux faits sont stockés dans la base de données implicite.

L'objectif majeur de ce travail est de développer un système où l'on pourrait déduire de nouveaux faits, ceux-ci étant implicites à partir des données de la base.

Ce système se nomme M.R.P.P.S 3.0. i.e Maryland Refutation Proof Procedure System, et est implanté sur un UNIVAC 1108.

Une relation est un ensemble de tuples dont les attributs appatiennent à des domaines différents.

Certains ont un nombre de tuples quasi infini et peuvent se définir de différentes manières. Elles sont appelées relations virtuelles (pour exemple: est_un_entier contient une infinité de tuples).

Les tuples des relations que l'on peut définir de manière explicite sont rangés dans la base de données explicite et sont exprimés sous forme de clauses singulières et complètement instanciées.

Les règles par lesquelles les relations virtuelles peuvent être définies sont rangées sous forme de clauses dans la base de données implicite. Ces dernières peuvent contenir des variables et des constantes. Le système MRPPS 3.0 est un système déductif qui relie ainsi les bases de données relationnellles et les systèmes experts.

Il possède un langage d'interrogation, un mécanisme déductif, une structure d'index pour accéder à des relations réelles ou virtuelles. il peut fournir des réponses en langage naturel ainsi qu'expliquer le raisonnement trouvé d'une manière écrite et orale.

Voici le schéma simplifié du système:

ici schéma (p28)

Les relations réelles et virtuelles sont entrées dans le bloc 1, leur syntaxe est ensuite vérifiée par un module testant la validité des formules bien formées. Si ces formules sont valides, elles sont stockées respectivement dans la base explicite et dans la base implicite, dans le cas contraire elles sont rejetées avec un message d'erreur à l'intention de l'utilisateur.

Les requetes de l'utilisateur doivent être mises sous forme de clauses, puis sont contrôlées dans le module de vérification, le bloc 3. Celui-ci compare les expressions au réseau sémantique (incluant le modèle de connaissance) contenu dans le bloc 6.

Le module de recherche déductive contrôle la recherche d'une réponse à une question. Certaines requêtes peuvent utiliser des éléments explicites et des éléments virtuels (stockés dans la base implicite).

Toutes les connaissances du système sont stockées dans le réseau sémantique du bloc 6 i.e les relations réelles, les définitions des relations virtuelles avec, de plus, des règles de contrôle des entrées et des informations sémantiques.

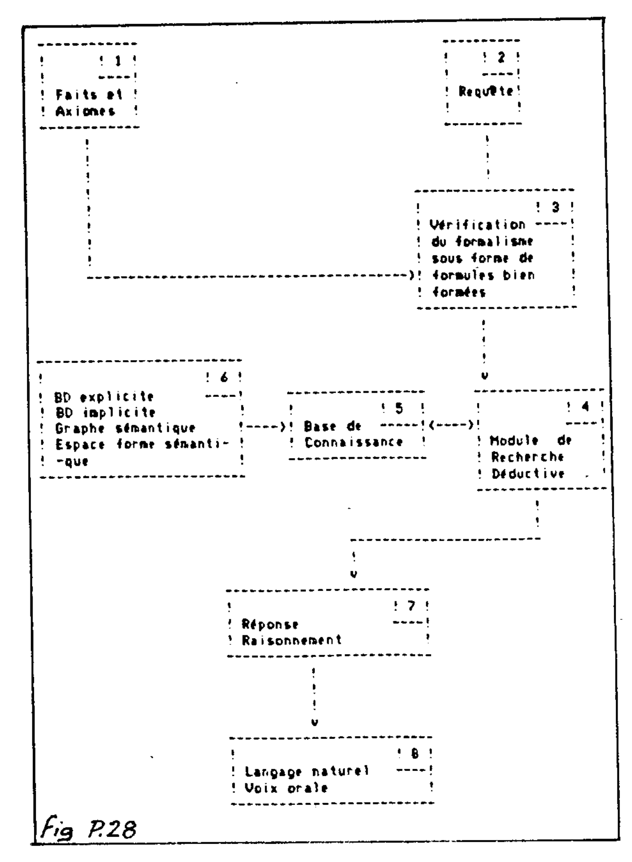


Fig P.28

Le module index de la base de connaissances permet un accès rapide aux bases implicite et explicite et a un espace de formes sémantiques définissant les règles de contrôle des entrées. Le mécanisme de déduction se termine quand une réponse à la question est trouvée. Les réponses et le raisonnement peuvent être donnés sous trois formes: la forme symbolique, le langage naturel écrit et le langage naturel oral. La réponse symbolique est sous la forme d'un arbre réponse/raisonnement.

Le module du réseau sémantique contient quatres éléments impotants:

. Le graphe sémantique qui spécifie les liens entre catégories

. La base de données rassemblant les données explicites (EDB) et les données implicites (IDB).

. Le dictionnaire des constantes, fonctions et catégories du système.

. L'espace de formes sémantiques définissant les contraintes sémantiques sur les valeurs des relations.

En logique multi-sorte, l'univers est composé de différents domaine. Chacun d'eux est appelé catégorie sémantique (pour exemple: zone géographique, un pays). Les relations entre ces catégories sont décrites dans un graphe, le graphe sémantique composé d'inclusions, de différences, de chevauchements entre les domaines (intersections). L'ensemble de tous les éléments d'une catégorie particulière représente une relation unaire.

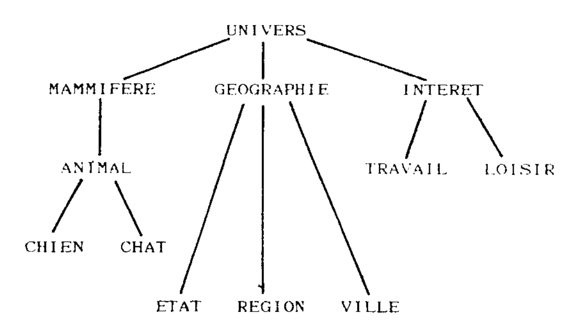
→

Exemple:

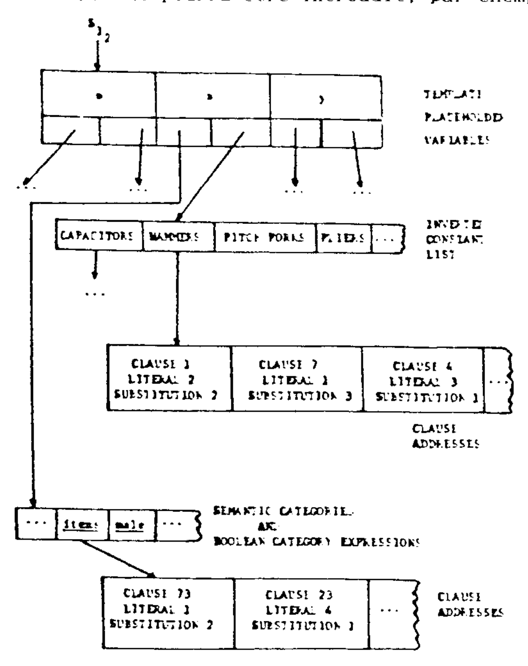
DOG(x) --> ANIMAL(x);ANIMAL(y) --> MAMMAL(y)
DOG(Fido)

Pour prouver que Fido est un mammifère, le raisonnement suivant est effectué: Fido trouvé dans le dictionnaire, pointe dans le graphe sur DOG, or DOG pointe sur ANIMAL qui pointe sur MAMMAL.

Ce qui nous donne le graphe suivant:



La base de données est sous forme de clauses, soit entièrement instanciées pour les données explicites, soit contenant des variables pour les données implicites. Le dictionnaire indique la catégorie sémantique de chaque relation constante ou fonction du système, exemple: nom de relation, élément de tuple. L'espace forme sémantique définit les contraintes que l'on impose aux prédicats et à leurs arguments, on inclut les containtes d'intégrités dans cet ensemble. La base de connaissances est formulée sous la forme d'un arbre qui contient des listes de constantes et des listes de catégories sémantiques. A chaque élément de ces listes est associé l'ensemble des clauses, littéraux substitutions où l'élément pourra être introduit, par exemple:



Les littéraux sont représentés sous forme d'arbre, le raisonnement de déduction utilise le "pattern-matching" adapté de l'algorithme d'unification de ROBINSON comme dans les Systèmes Experts. Le mécanisme de déduction est basé sur le principe de déduire une nouvelle clause à partir de deux clauses données.

M.R.P.P.S 3.0 est un système à réfutation de preuve, il réfute le problème à résoudre et essaie de trouver une contradiction.

Exemple:

à l'aide des relations: fabrique, utilise, fournit

Fabrique	Nom du fabricant	Produit	Région
	ABC	Marteau	Maryland
	ABC	Fourchette	Maryland
	ACME	Résistance	Virginie
	ACME	Capacité	Virginie
	ACME	Diode	Virginie

Utilise	Compagnie	Article utilisé	Région
	Radio	Capacité	New York
	Radio	Résistance	New York

En outre, une relation virtuelle est définie dans l'IDB:

fabrique(u,v,w) et utilise(r,v,s) -->
fournit(u,r,v)

La question est (3x) fournit(ACME,RADIO,x) ?

C'est à dire: qu'est ce que fournit la compagine ACME à la compagnie RADIO.

La réponse du système par voix orale est la suivante:

1 réponse
l'entreprise ACME fournit des capacités, des résistances à l'entreprise RADIO

2 règle générale
puisque l'entreprise ACME fabrique des capacités, résistances en VIRGINIE et que l'entreprise RADIO utilise des capacités et des résistances à NY, on peut conclure que l'entreprise ACME fournit des capacités et des résistances à l'entreprise RADIO.

3 faits
l'organisation RADIO est située à NY et utilise des capacités et des résistances.

4 fait
l'organisation ACME est située en VIRGINIE et fabrique des capacités et des résistances.

Le processeur vocal de MRPPS 3.0 est une amélioration de celui développé au laboratoire de Recherches Naval en 1975.

Ce système est une bonne approche du problème, néanmoins il ne permet pas de réutiliser les SGBD existants. D'autre part, le système utilise le modèle relatinnel transcrit sous forme de résaux sémantiques, de plus, les requêtes sont énpncées sous la forme de formules du calcul des prédicats ce qui rend l'interface utilisateur peu conviviale.

DEDUCTIVELY AUGMENT DATA MANAGEMENT
(KELLOG, KLAR, TRAVIS)

D.A.D.M. est une base de données déductive, conçue par KELLOG, KLAR et TRAVIS à l'Université du WISCONSIN. D.A.D.M. consiste à enrichir d'un module déductif un Système de Gestion de Base de Données Relationnelle classique.

Des techniques de plan d'inférence ont été incorporées à un processeur déductif dont l'objectif est de d'extaire des informations implicites à partir du contenu d'une base de données relationnelle. Chemin déductif et plan d'inférence sont utilisés pour sélectionner de petits ensembles de prémisses pertinentes, pour construire des bases de déduction. Lorsque ces bases sont vérifiées, le système les utilise comme plan pour créer des stratégies d'accès à la base de données afin de guider la recherche des valeurs, l'assemblage des réponses et la production des preuves soutenant ces réponses.

Ainsi, DADM tenre de résoudre un inconvénient des SGBD actuels, l'incapacité de découvrir des relations implicites à partir de celles présentes dans la base de données. Ce prototype a deux objectifs:

1- Permettre à l'utilisateur de poser des requêtes complexes au système, qui se chargera de trouver des connexions entre les concepts spécifiés par l'utilisateur et les structures de la base de données.

2- Générer pour l'utilisateur des informations dérivées de la base de données.

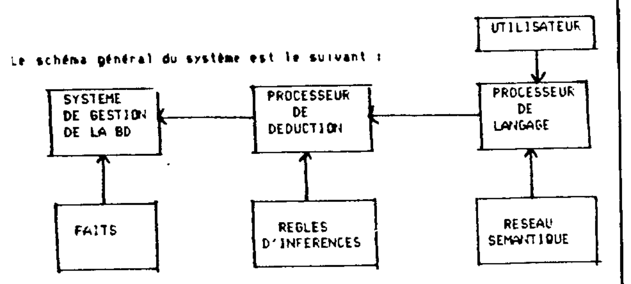
Pour cela, des techniques interactives entre l'utilisateur et le système ont été développées afin que le système crée et affiche des plans et des chaînes d'évidences. L'utilisateur participe donc activement en donnant des conseils ainsi qu'en raffinant ses questions. ce cycle interactif continue jusqu'à ce que l'utilisateur soit satisfait tant de la quantité, que de la qualité des informations dérivées. Parfois, le système fournit une réponse conditionnelle plutôt qu'une réponse catégorique. dans tous les cas, le système demande confirmation d'un résultat en questionnant l'utilisateur.

Le système est constitué d'un processeur de déduction basé sur les réseaux sémantiques et d'un module d'exploitation d'un ebase de données relationnelle. L'analyse d'une requeète est effectuée par la comparaison avec le réseau sémantique. Le processeur de déduction est indépendant du SGBD, son ajout ne nécessite donc aucune modification de celui-ci.

Nous pouvons distingeur, d'une part le SGBD qui gère les données explicites, et d'autre part, le processeur de déduction qui utilise des données implicites (règles d'inférence, axiomes).

Les requêtes ne nécessitant pas de recherche déductive sont envoyées directement au module de gestion de base de données. Le processeur déductif sélectionne les prémisses des règles pertinentes et fabrique des plans d'inférence (plan de déduction); il guide de plus la recherche dans la base de données des faits nécessaires à la déduction.

Le schéma du système est le suivant:



Les règles et requêtes sont entrées dans le système sous l'aspect d'expressions en forme normale du calcul des prédicats du premier ordre, avec le signe d'implication. Dans les règles ainsi formées, les hypothèses des questions sont situées à gauche de l'implication alors que les buts sont à droite. Chaque prédicat est une relation de la base. Cette formulation des connaissances est ainsi la conjugaispn du modèle relationnel avec des modèles utilisés dans les systèmes experts.

Plusieurs sortes d'informations sont extraites des règles à l'entrée et sont utilisées pour creer un graphe de connexion des prédicats (GCP). L'implication à l'intérieur d'une règle est représentée par un lien de dépendance dans ce graphe. Les liens de nouvelles règles avec d'anciennes sont décrits à l'aide d'arcs. ■

Suite de la page 6

Un programme écrit sur IBM 3090 tournera sur QL-SINCLAIR (dans les limites imposées par le matériel bien sûr); les difficultés d'adaptation ne viendront pas de l'APL, mais de la disparité des systèmes et des supports.

UN LANGAGE INTERNATIONAL

Les opérateurs des primitives APL sont représentés par des symboles dont beaucoup sont des symboles mathématiques classiques:

+ - ÷ ! x etc...

Cet aspect de l'APL rebute certains débutants bien que pratiquemment la mémorisation des nouveaux symboles se fasse progressivement et sans douleur. L'APL est indépendant de toute langue nationale: on n'y retrouve pas de mots clés tels que GOTO, WEND, ELSEIF etc... qui font ressembler certains langages à du bishlamar (sorte de sabir franco-anglais parlé dans certaines iles du Pacifique).

Pour exemple, on veut remplacer la division " " par une fonction qui effectue la même opération, mais qui, si le diviseur est nul, renvoie la valeur du dividende au lieu d'émettre un message d'erreur. On peut écrire une fonction qu'on appellera "DIVISER" ou "DIVIDE" selon la nationalité de chacun et qui prendra en compte le cas particulier du diviseur nul. Au lieu d'écrire:

A B
on écrira
A DIVISER B

On aura créé un mot-clé qui permettra de faire certaines opérations que l'on aura défini en utilisant la même syntaxe que pour les opérateurs de base. Dans la définition du mot-clé (ou fonction) on peut utiliser les opérateurs de base ainsi que les mots-clés (fonctions) préalablement définis. C'est là l'aspect évolutif du langage APL.

UN LANGAGE MODULAIRE

Un progiciel APL bien conçu est un ensemble modulaire et structuré de fonctions élémentaires à l'intérieur desquelles tous les types de branchement sont possibles. Cette organisation modulaire permet une maintenance aisée, contrairement à une légende complaisamment propagée.

Cette possibilité de structuration à l'échelle macroscopique et d'utilisation des branchements à l'échelle microscopique procure au programmeur APL un outil plus souple et plus puissant que ne le sont les langages structurés par nature et par doctrine.

APrès ces précisions nécessaires mais un peu indigestes, nous verrons par la suite comment utiliser concrètement un interpréteur (ou interprête) APL. ■

Complément du n° 27

SINUS DE PRECISION

A partir d'une table au pas de 10°, un calcul d'erreur montre que, jusqu'à 5 décimales, la séquence suivante convient:

```
   x' = angle en radian ( < 0,088 )
   u' = x'**2
                       .     .    .         .
V' = (( 1 + u'/3 ) * x' * C' + S' ) * (1-u'/2)
                  T'x  B'   D'   E'          V'
```

Pour une machine 16 bits, le problème est celui de la troncature. Pour cela,
- toutes les valeurs sont multipliées par un module M inférieur à 65300.
- la multiplication devient: M */
- u' est représenté par u = 64 * M * u'
- les valeurs sont arrondies avant remise à l'échelle
- l'arrondi est compensé en majorant Tx de 1 .

Le programme devient:

```
( SCx-- )
↑8*↑M*/R!32+192/M+1+M*/ M*/ + ↑R> 128/ M*/ -
      u            Tx  B   D E               v
```

Dans le cas présent, les diagrammes sont avantagesement remplacés par une disposition judicieuse de la formule algébrique, et des commentaires.

La programmation des piles est un peu plus difficile que la simple écriture d'une formule algébrique, mais les méthodes montrées dans cet article devraient permettre sans problème une programmation optimale, c'est à dire:
exacte, rapide d'écriture, mise au point, et exécution avec une bonne documentation.

LA PROGRAMMATION STRUCTUREE

Pourquoi la programmation structurée, alors que les premiers langages ne le permettaient pas?

Ils étaient axés vers le fonctionnement de la machine, qui ne savait, et ne sait toujours faire que:
- **transférer un mot**
- **le mémoriser**
- **soustraire, avec ou sans retenue**
- **se brancher à une adresse donnée.**

Ce qui avait deux inconvenients:
- **non portabilité: un programme était très coûteux à traduire pour une autre machine.**
- **quasi impossibilité de démontrer qu' unprogramme correspondait à un algorithme donné.**

Le programme est devenu l'expression d'un algorithme à base de DEBUT FIN SI SINON TANT-QUE et non plus une liste d'ordres.

Néanmoins, le GOTO n'a pas entièrement disparu, sauf de quelques langages comme le FORTH.

Comment faire alors pour retranscrire un programme dont l'organigramme semble fait par un cuisinier italien qui aurait oublié ses ciseaux à spaghetti?

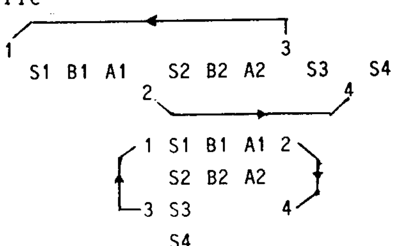
Il y a plusieurs façons de s'en sortir:
- **quitter le bloc logique grâce à des instructions comme LEAVE du FORTH ou BREAK du langage C.**
- **quitter le sous-programme par RETURN ou EXIT.**
- **faire un calcul de booléen avant chaque aiguillage.**

Quelques exemples pour illustrer ceci:

NOTATIONS

```
?(  IF     -{ BEGIN     A Aiguillage
()  ELSE   (  WHILE     B Booléen
)-  THEN   )) REAPEATS  Séquence d'instructions.
           ?) UNTIL
CONVENTION:  BOOLEEN: 0: Branchement
                      autre: poursuite en séquence.
```

L'organigramme suivant peut être présenté de façon horizontale ou verticale:



Le branchement avant doit être remplacé par deux aiguillages IF THEN de part et d'autre du point 3, et le booléen B2 changé en:

B3 B1 NOT OR ⟶ B'_2

Le programme FORTH devient alors:

```
BEGIN   S1 B1           | -{   S1 B1
  IF    S2              |  ?(  S2
  THEN     B2 B1 NOT OR |  )-      B2 B1 NOT OR
UNTIL      B1           | ?)       B1
IF      S3              | ?(   S3
THEN    S4              | )-   S5
```

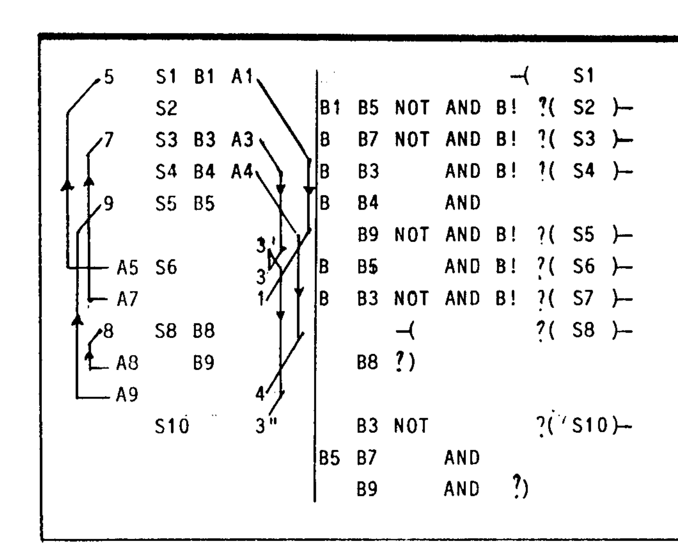
Voici un deuxième exemple plus complexe. Deux solutions sont possibles:
1. **) calculer le booléen avant chaque aiguillage**
2. **) actualiser une variable booléenne avec les mots AND et NOT OR avant chaque aiguillage: ceci a l'avantage de permettre aux booléens élémentaires de varier après l'instant de prise en compte logique.**

C'est la méthode retenue ici:
- **une variable BOOLEAN est crée, ainsi que les mots FORTH:**

```
: B BOOLEAN   ;      : B! DUP BOOLEAN ! ;
```

- **ne sont conservées que les boucles BEGIN UNTIL imbriquées.**

→



Ces exemples, surtout le deuxième, tiré d'un programme optimisé en assembleur, ne seraient pas très efficaces: pour limiter les aiguillages dans les boucles BEGIN UNTIL, il vaut mieux reprendre l'analyse. Souvent, il suffit de faire des pré ou post incrémentations.

Pas d'affollement, donc, devant un organigramme en plat de spaghetti: il suffit d'un peu d'algèbre de BOOLE pour avoir la solution.

FORTH EXPERTISE-COMPLEMENT AU N°25

par J.M. PREMESNIL

Dans un précédent numéro, nous diffusions une application programmée à l'aide du logiciel EXPERT-2 appliquée au bâtiment. Nous avons omis un schéma et deux tableaux. Nous réparons donc ici cet oubli.

PLUIE BATTANTE
H3
EAUX DE RUISSELLEMENT
H2
S/SOL
NAPPE PHREATIQUE
H1

	H1	H2	H3	H4	H5	H6	H7
-Eau dans cave	X						
-Cave humide	X	X		X			
-Murs cave ruisselant l' hiver	X			X			
-Idem mais en étage courant				X			
-Plinthes pourries au rez-de-chaussée	X	X					
-Papier peint ou peinture cloquée en bas des murs	X						
-Idem mais en haut des murs							X
-Salpetre sur mur	X		X	X			
-Taches de moisissure sur exterieur bas mur périphérique		X					
-Idem en façade toute hauteur			X			X	
-Idem sur cloisons intérieures					X		
-Joints de carrelage boursouflés au rez-de-chaussée	X						
-Plafond taché au centre en étage sous comble						X	
-Idem mais en coin ou en périphérie de mur de façade							X
-Idem mais en étage courant					X		
-Décollement de carrelage localisé dans salle de bain					X		
-Idem mais général				X			
-Sensation humide dans pièce chauffée				X			
-Idem dans pièce froide	X	X					X
-Buée insistante dans salle de bain ou cuisine				X			
-Taches sur plancher du grenier							X
-Taches localisée en périphérie du grenier							X
-Taches au pied des façades extérieures	X	X					
-Taches localisées en façade			X				
-Tache en partie haute de la façade						X	

TABLEAU 1

LOCALISATION DE L'HUMIDITE	H1	H2	H3	H4	H5	H6	H7
1 Partie basse du batiment	X	X					
2 Façades (faces intérieures ou extérieures)	X	X	X	X			X
3 Intérieur du batiment sauf façades	X			X	X		
4 Partie haute du batiment						X	X

TABLEAU 2

FORTH FONCTION SINUS : TRACE D'ARC DE CERCLE

par M. PETREMANN

THOMSON

```
VARIABLE X0   VARIABLE Y0


: CENTRE   ( X0 Y0 --- )
199 SWAP  - Y0 ! X0 ! ;

: POSX1Y1   ( r deg --- X1 Y1 )
  OVER OVER *COS X0 @ +
  ROT  ROT  *SIN Y0 @ SWAP - ;

: ARC   ( r DEB FIN --- )
1+ >R  OVER OVER
POSX1Y1  PSET  R>
SWAP
DO
  DUP  I  POSX1Y1  LINETO
6 +LOOP
DROP ;
```

Le mot CENTRE initialise les valeurs XØ et YØ avec les valeurs déposées sur la pile. Attention, XØ et YØ correspondent aux coordonnées du centre du cercle à définir, position définie par rapport à l'origine des axes Ox Oy dans un repère trigonométrique. Ce repère ne correspond pas au repère des coordonnées graphique du TO7. En effet, ce repère commence en haut et à gauche de l'écran, c'est pourquoi l'opération 199 SWAP - située dans CENTRE permet de replacer l'origine en bas et à gauche de l'écran du moniteur.

Le mot POSX1Y1 calcule les valeurs X1 et Y1 correspondant à la position du point du cercle à tracer. Cette position se calcule comme suit:

```
Y1=YØ + r * sin(a)
X1=XØ + r * cos(a)
```

Le mot ARC trace un arc de cercle de rayon r et se traçant de l'angle DEB à l'angle FIN. Ainsi, pour tracer un cercle de rayon 5Ø dont le centre est aux coordonnées XØ=12Ø et YØ=15Ø, il faut taper:

```
12Ø  15Ø  CENTRE  5Ø  Ø  36Ø  ARC
```

Dans la définition de ARC, le mot PSET positionne un point aux coordonnées x y, où x est le nombre de pixels à partir de la gauche de l'écran et y le nombre de pixels à partir du haut de l'écran. Le mot LINETO trace un trait à partir du dernier point tracé (par PSET ou LINETO) vers le point de coordonnées absolues x y.



RESUME DES COMMANDES D'EDITION

	ED		Edite l'écran courant.
n	EDIT		Edite l'écran n.
n	LIST		Affiche sur le terminal l'écran n.
	L		Affiche sur le terminal l'écran courant.
	A		Pointe l'écran commentaire.
n	N		Appelle comme courant l'écran de numéro suivant.
n	B		Appelle comme courant l'écran de numéro précédent.
	TOP		Place le curseur ligne 0 colonne 0.
n	+T		Avance le curseur n lignes plus loin.
n	T		Place le curseur ligne n colonne 0.
n	C		Déplace le curseur de n caractères.
	I	txt	Insère le texte à la suite du curseur.
	O	txt	Surimpressionne le texte à la suite du curseur.
	P	txt	Remplace le texte sur la ligne courante.
n	NEW	txt	Remplace le texte sur la ligne n et continue sur la ligne suivante jusqu'à l'entrée d'un texte vide.
	F	txt	Cherche le texte et place le curseur à sa suite.
n	S	txt	Cherche le texte jusqu'à l'écran n et place le curseur à sa suite.
	R	txt	Remplace le texte trouvé par F ou S.
	E		Efface le texte trouvé par F ou S.
	D	txt	Cherche le texte et l'efface.
	U	txt	Insère une ligne blanche à l'emplacement du curseur.
	TILL	txt	Efface du curseur jusqu'au texte compris.
	JUST	txt	Efface du curseur jusqu'au texte non compris.
	KT	txt	Efface du curseur jusqu'au texte compris en le plaçant dans le tampon ''INSERT'.
	X		Efface la ligne courante et la déplace dans le tampon ''INSERT'.
	WIPE		Efface l'écran en totalité.
	K		Echange le contenu des tampons INSERT et FIND.
	SPLIT		Déplace le reste de la ligne sur la ligne suivante.
	JOIN		Regroupe la ligne suivante à la ligne du curseur.
el	G		Copie la ligne l de l'écran e sur la ligne courante.
elm	BRING		Copie les lignes de l à m de l'écran e à partir de la ligne courante.
	W		Sauvegarde sur disque les écrans modifiés.
	QUIT		Sortie éditeur sans sauvegarde écrans modifiés.
	DONE		Sortie éditeur en sauvegardant les écrans modifiés.

```
HEAT         !
FALCO        !
TELEVIDEO    !      Appels aux commandes
QUME         !   de terminal vidéo particulier.
ANSI         !
PERKIN       !
DUMB         !
```